Martin Eisemann; Thorsten Grosch; Marcus Magnor; Stefan Müller

Automatic Creation of Object Hierarchies for Ray Tracing of Dynamic Scenes

Auch erschienen als:
Technical Report 2006-6-1. - Computer Graphics Lab, TU Braunschweig

Martin Eisemann
eisemann@cg.tu-bs.de

Thorsten Grosch
grosch@uni-koblenz.de

Marcus Magnor
magnor@cg.tu-bs.de

Stefan Müller
stefan.mueller@uni-koblenz.de

Automatic Creation of Object Hierarchies for Ray Tracing of Dynamic Scenes

Technical Report 2006-6-1

June, 12, 2006

Computer Graphics Lab, TU Braunschweig

Abstract

Ray tracing acceleration techniques most often consider only static scenes, neglecting the processing time needed to build the acceleration data structure. With the development of interactive ray tracing systems, this reconstruction time becomes a serious bottleneck if concerned with dynamic scenes. In this report, we describe two strategies for efficient updating of bounding volume hierarchies (BVH) for scenarios with arbitrarily moving objects. The first exploits spatial locality in the object distribution for faster reinsertion of the moved objects. The second allows insertion and deletion of objects at almost constant time by using a hybrid system, which combines benefits from both spatial subdivision and BVHs. Depending on the number of moving objects, our algorithms adjust a dynamic BVH six to one hundred times faster than it would take to rebuild the complete hierarchy, while rendering times of the resulting hierarchy remains almost untouched.

1 Introduction

Ray tracing is well known for its ability to create photorealistic images. However, it is usually considered to be too slow for interactive applications. Recently developed ray tracing systems however are able to achieve interactive frame rates, but their efficiency relies heavily on precalculated acceleration data structures [14][19][3][16]. The complexity for reconstructing these acceleration data structures for a scene with n triangles is often $O(n \log n)$ or worse, with the final cost in the ray tracing phase being only $O(\log n)$ per pixel on average. This usually limits interactive ray tracing to static scenes or simple walkthroughs, so that the acceleration structure can be reused for all frames.

For a complete interactive ray tracing system, an efficient support of moving objects is necessary. Therefore, the acceleration data structures usually have to be rebuild for each frame. Techniques like frameless rendering [1] [2] and frustum traversal [16] reduce the amount of work that has to be done in the ray tracing phase and almost linear scalability for up to 128 processors has been shown [14]. But the reconstruction phase can not be parallelized as easily, in fact very little research has been done on this topic [7]. Thus it becomes the bottleneck to interactive ray tracing of dynamic scenes.

In this article we present two approaches to deal with the problem of ray tracing animated scenes, based on bounding volume hierarchies (BVH). The first, called *Dynamic Goldsmith and Salmon* (Dyn. G&S), exploits spatial coherence to rapidly update the BVH, while the second, called *Loose Bounding Volume Hierarchy* (LBVH), is a hybrid approach, which allows for reconstruction of the acceleration data structure in $O(n)$ by combining the benefits of a BVH with spatial subdivision.

The rest of the report is organized as follows. In the next section we review some related work. Then we present our approaches of handling dynamic scenes in Sect. 3. Results and their discussion are given in Sect. 4, followed by a conclusion and directions to future work in the final section.

2 Related Work

A large number of methods and algorithms to speed up ray tracing exist, but almost all of them are designed for static images or simple walkthroughs and not much attention has been spent on constructing these acceleration structures in an efficient manner. Therefore, ray tracing of dynamic scenes is a rather new field of research, which gets more and more important as ray tracing gets more and more accelerated.

Quite early Glassner developed a technique called *Spacetime Ray Tracing* [4]. The idea was to intersect rays with static four-dimensional objects

instead of dynamic objects in three-space, whereas the fourth dimension is time. Unfortunately, this technique is only suitable for scenes with predefined movements.

Using multiprocessors Parker et al. [14] were able to ray trace reasonably complex scenes at interactive frame rates. Moving objects are tested separately for intersection, which therefore allowed only a small amount of them (≤ 10).

Reinhard et al. [15] used hierarchical grids for ray tracing of dynamic scenes. Their data structure is essentially a balanced octree, which keeps objects at different levels, depending on their size. This allows for insertion and deletion in almost $O(1)$ for an object. Depending on the motion, the entire data structure needs to be rebuild once in a while.

Lext and Akenine-Moeller [11] build hierarchies of oriented bounding boxes containing recursive grids. These grids include all primitives which underly the same affine transformation, it is therefore sufficient to build them once and transform the rays into the local coordinate system for performing intersectiontests.

Wald et al. also exploit local coordinate systems to animate rigid bodies [20]. But instead of using the scene graph for traversal between these entities, they rebuild the whole top-level data structure every time a movement takes place.

Guenther et al. use motion decomposition to ray trace deformable models, whose connectivity does not change and for which the space of possible poses is known in advance, by decomposing a model into clusters which underly a similar transformation [6]. Residual motion is captured in a single fuzzy kd-tree for the entire animation.

An example of a lazy evaluation strategy was given by McNeill et al. [13]. The upper levels of an octree are build in a preprocessing step, while the rest is build on demand. They propose to use this technique also for dynamic environments, but test results were only presented for static scenes.

Actually, Larsson and Akenine-Moeller [9] make strong use of lazy evaluation to ray trace deformable models, by utilizing the static connectivity between the triangles and refitting only the upper half of their preconstructed BVH. The rest gets refitted on demand. As the structure of the BVH is not allowed to change, the possible movement of the triangles is rather limited without degrading performance, even though it can be sufficient for ray tracing small to mid-sized deformable scenes [21].

To make this technique applicable for any kind of scenes, Lauterbach et al. used the ratio between each parent node's surface area to the sum of the area of its two children to detect degradation of the BVH and rebuild it on demand [8].

A quite interesting approach was also presented by Ulrich [17], called *Loose Octrees*. It has not been used in the context of ray tracing so far, only for collision detection and view frustum culling. These Loose Octrees

are a variation of normal octrees which allow insertion in $O(1)$ by using overlapping voxels and choosing an insertion level depending on the size of the object. But this overlapping is also the reason why the scheme works better for collision detection than for view frustum culling.

3 Our Approaches

Rebuilding a BVH for every frame of an animation, using standard techniques, make it impossible to achieve interactive frame rates in rather complex scenes. A refitting of the bounding volumes (BV), a recomputing of the bounds of the BVs, can be done very quickly. But, depending on the movement of the objects, the quality of the BVH can arbitrarily decrease, resulting in inacceptable long rendering times for certain scenes.

In this report we present two approaches to deal with ray tracing of dynamic scenes with non-deterministic movement. In Sect. 3.1 a method is presented, which exploits locality in the acceleration structure for a rapid update. The second approach in Sect. 3.2 presents a method for insertion and deletion of an object into a BVH in almost constant time. For both methods we introduce a new phase between each frame, the *update phase*, in which the animated objects are moved and the update of the BVH takes place.

For an easier understanding, we first clarify some terms. A *primitive* can be any kind of basic geometric shape, like a triangle, a parametric shape, and so on. An *object* on the other hand can be either a primitive or a collection of primitives within its own local coordinate system having its own acceleration structure, in our case a BVH. These objects are stored in a separate list. All leaf cells of our BVHs possess a pointer towards the object contained in them. These nodes are called *object nodes*. In addition, the objects have a *hierarchy pointer*, if necessary, which grants immediate access to the object node in the BVH containing the object. This is actually one of the biggest advantages of BVHs compared to other acceleration structures, since all objects are contained in just one single node of the BVH, instead of several voxels, as it could be possible when using k-d trees, octrees or uniform grids. The relation of these terms is given in Fig. 1.

3.1 Dynamic Goldsmith and Salmon

In this section we describe an adaptive hierarchy, based on the BVH creation scheme introduced by Goldsmith and Salmon [5], which we used to initially build the BVH, even though the described technique is not limited to this creation scheme and others, like the surface area heuristic could be applied as well [12]. Which scheme suits best is unfortunately always scene dependend.

Goldsmith and Salmon proposed one of the earliest methods to deal with dynamic changes in a scene, by deleting the object nodes from a BVH, ad-

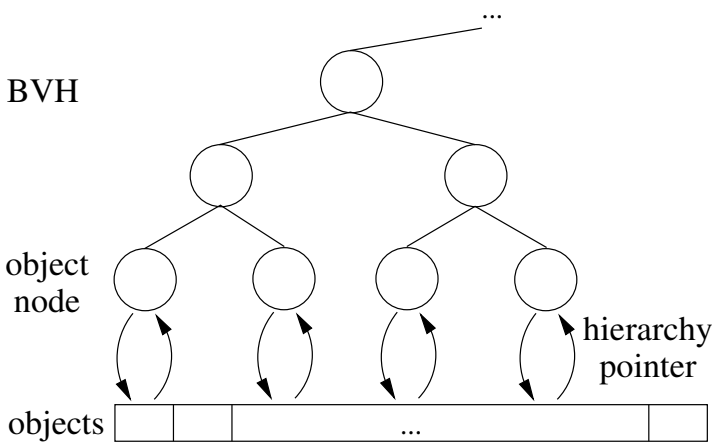

Figure 1: Overview of the notations. BVH: Bounding Volume Hierarchy

justing the BVs and reinserting the object nodes beginning at the root, using a heuristic tree search to find the optimal insertion position (for more details see [5]). However, this technique is a rather inefficient scheme. Therefore we changed that approach. It is not necessary to completely remove a changed object node from a BVH. Since a certain spatial locality is given by the BVs in the same subtree of a BVH and since objects usually move only small distances compared to the scene extent, the object would either be inserted at its old position in the BVH or a position nearby for the most part. The term nearby here means a subtree which encloses both, the old and the new position of the object. Since this subtree is probably much smaller in depth compared to the whole BVH, starting the reinsertion of the object node at the root of this subtree will shorten the whole insertion process drastically. We will call the root of this subtree the *reinsertion node*. Choosing this node, we minimize the number of needed insertion steps, since no changes have to be made to any of its ancestors. This also implies that an insertion starting at the root would most likely lead to this node anyway. This is depicted in Fig. 2. As long as object A stays inside the bounds of the dashed BV, it will most likely be reinserted in the corresponding subtree. In the worst case, this process is exactly the same as removing an object node completely from the BVH and reinserting it at the root node. In comparison to a complete rebuild this can speed up the process of reconstructing the hierarchy by more than two orders of magnitude (see the test results in sect. 4).

This process is also visualized in Fig. 3. From its old position, the object node is passed along its parents until the reinsertion node is found, which is the first node on the path to enclose the object node, and is inserted again, leading to its new position.

Removing an object from a BVH and reinserting it may lead to an ef-

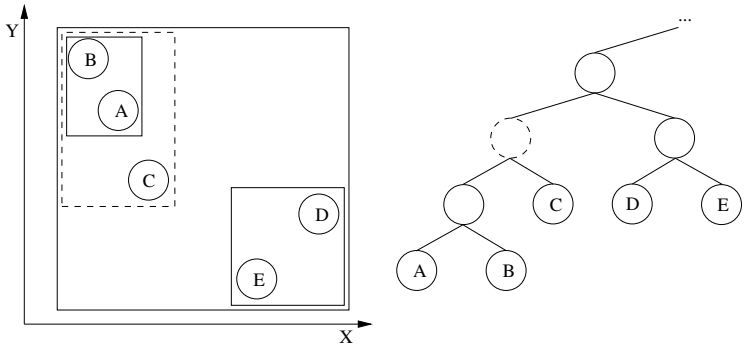

Figure 2: Reinsertion at inner nodes of a BVH. As long as moving object A stays inside the bounds of the dashed BV it will most likely be reinserted somewhere in the corresponding dashed subtree shown on the right.

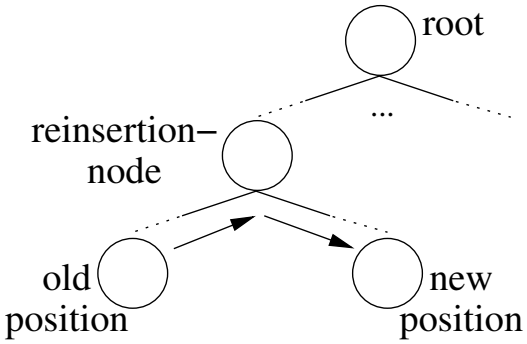

Figure 3: Beginning at its old position, the object node of the moving object is passed along its ancestors, until the reinsertion node is found, from where it is inserted again.

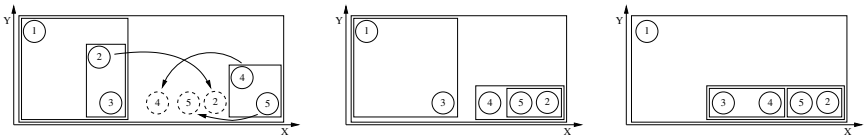

Figure 4: Example for the thinning of nodes. Left: The objects move as depicted by the arrows. Middle: resulting BVs after movement. Right: Rearranged BVH for a faster traversal.

fect which we will call *thinning*. This term describes a decrease of objects contained in a node, while its surface area remains almost unchanged. If the thinning continues, it is most likely that better BVHs could and should be created. An example is given in Fig. 4. Objects 2, 4 and 5 are moving as depicted by the arrows. The dashed circles are the target positions (see Fig. 4 on the left). At a certain point in time the insertion criteria would force object 2 to change into the right subtree (Fig. 4 in the middle). Even though object 3 did not move at all, it would result in a better BVH if it would change into the right subtree as well (Fig. 4 on the right).

To prevent such a degradation of a BVH, we introduce a quality criterion $Q(B)$ that can be efficiently calculated and effectively prevented thinning in our tests. We use the surface area of a node divided by the number of objects contained in its subtree, which is in some sense a measure for the packing density of this node. This is depicted in equation (1):

$$Q(B) = \frac{S(B)}{C_{obj}(B)} \quad , \tag{1}$$

where $Q(B)$ is the quality measurement of node B, $S(B)$ is the surface area of B and $C_{obj}(B)$ is the number of objects contained in the corresponding subtree of B.

After the initial construction of the BVH, an initial value Q_{init} is calculated for every node in the hierarchy. This is also done during the update phase, if a new node is created. During the animation, the current value $Q_{current}$ of a node is compared to its initial value. If it exceeds a predefined threshold, it gets deleted and the children of the node are reinserted as described above.

In our tests this quality criterion not only removes most of the threat coming from thinning of nodes, but can also decrease the ray tracing phase up to 34%, while only increasing the update time by about 16% compared to not dealing with thinned nodes. Of course these values are scene dependend.

Combining both presented techniques leads to the following pseudocode:

The underlying data structure is highly dynamic, which means keeping a good cache efficieny is non trivial. For some high performance systems it might be a better solution to just mark all reinsertion nodes and rebuild the whole underlying part of the BVH. Since every node has a fixed memory

Algorithm 1 Update Phase Dyn. G&S

```
1: for all objs do
2:    animate objs
3: end for
4: for all animated objs do
5:    adjust BV of obj node
6:    remove obj node from hierarchy
7:    find insertion node
8:    insert obj node
9: end for
10: remove thinned nodes
```

footprint, this reconstruction can be done in place, this memory bound is not available for other spatial data structures, like kd-trees.

3.2 Loose Bounding Volume Hierarchy

Even though the method described in the last section can result in a tremendous speed-up to the update phase, its complexity is still no better than $O(m \log n)$ on average given m moving objects and n scene objects. In the following we present a hybrid approach, which allows insertion and deletion of objects in $O(1)$ by exploiting that every object lies exactly in one node of a BVH combined with a pseudo-spatial subdivision scheme.

Using a pre-built BVH with a fixed subdivision level of $3N$ and a branching factor of $k = 2$, we can think of the lowest level as a uniform grid which encloses the whole scene and with a $2^N \times 2^N \times 2^N$ resolution. To define the insertion position for the object nodes, based on their midpoints, the index i_x of the voxel in the x-direction can be calculated using the following equation:

$$i_x = \left\lfloor 2^N \left(\frac{O_{x_{mid}} - S_{x_{min}}}{S_{x_{max}} - S_{x_{min}}} \right) \right\rfloor \quad , \tag{2}$$

where $O_{x_{mid}}$ is the midpoint of object node O along the x-axis, and $S_{x_{min}}$ and $S_{x_{max}}$ define the minimum and maximum value of the scene extent along this axis. Similar computations are made for i_y- and i_z-axis. The actual index in the BVH can then be computed from these indices, depending on the chosen memory layout.

The LBVH for a simple test scene is shown on the left in Fig. 5. Empty nodes in the graph are represented by dots. Dotted lines in the scene on the left represent extents of the voxels for insertion, bounding volumes are drawn with solid lines. Identical bounding volumes are drawn with different scales for clarity.

Since all objects are inserted at the lowest level, we can not assure that

the children of a node are actually smaller than its parent, which is a must have for a BVH with a reasonable performance. To solve this problem, we allow inner nodes to contain objects as well. For every object, the insertion level is calculated from its AABB using the following equation:

$$L = 3 \left\lfloor \log_2(\min(\frac{S_x}{O_x}, \frac{S_y}{O_y}, \frac{S_z}{O_z})) \right\rfloor \quad , \tag{3}$$

where O_a is the extent of the AABB of object O along axis $a \in \{x, y, z\}$ and S_a is the extent of the AABB surrounding the scene along axis a. Using equation (3) we keep larger objects closer to the root and therefore assure, that the maximum possible extent along the splitting axis is reduced by 50% for every level of the hierarchy, which leads to a good spatial partitioning. If L is greater than the predefined subdivision level of the BVH, it is set to the maximum possible level.

Depending on L we can calculate the indices in the x-, y- and z-direction similar to equation (2), substituting N for $L/3$. For the x-direction this is shown in equation (4).

$$i_x = \left\lfloor 2^{L/3}(\frac{O_{x_{mid}} - S_{x_{min}}}{S_{x_{max}} - S_{x_{min}}}) \right\rfloor \tag{4}$$

Because of the limited number of possible indices, the easiest way to calculate the index in the BVH is to use a precalculated lookup table, based on i_x, i_y, i_z and L. This also allows for any desired memory layout of the BVH, optimized for the chosen traversal method. The object node is then added to that node. The resulting BVH, using the same small test scene as before, is shown in Fig. 5 (middle). Object A will be assigned to the root node due to its great extent along the x-axis, while B stays at its old node.

If we assume constant scene extends, we could calculate the i_x, i_y and i_z even faster if we transform the whole scene into the $N \times N \times N$ volume, since the calculation of the index becomes a simple truncating of the midpoint coordinates in this volume.

The insertion process may lead to nodes with just one child. This can happen if small objects are surrounded by a large empty space. Because the object node and one or more of its ancestors are identical in this case, it would be a tedious task to test all of them for intersection. To avoid this problem *skip indices* can be used. If a ray intersects a node, usually all children have to be tested for intersection as well, but instead of testing the child directly, the node that its skip index points to is tested for intersection. This way all nodes with just one child and without objects can be skipped easily. An example for our simple test scene is shown on the right of Fig. 5. The calculation of the skip index can be efficiently done in the refitting process, which will be described in the following.

Until now objects are inserted into the hierarchy, but the BVH ist still inconsistent, since only the nodes containing at least one object could be

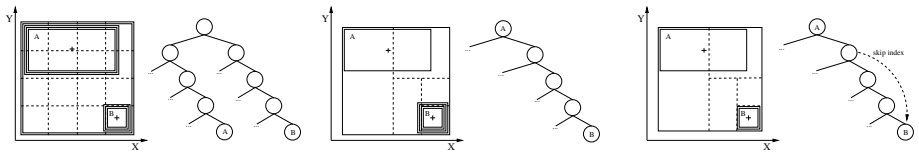

Figure 5: Left: Simple version of the Loose Bounding Volume Hierarchy, with objects inserted at the lowest level. Middle: Advanced version, keeping larger objects at higher levels. Right: Final version, using skip indices in addition to the advanced version to avoid intersecting unnecessary nodes.

adjusted so far. We can allocate the BVH in memory in a 1D array. We are able to do this, since the number of nodes is known a priori. Assuring that the index of a node is less than the index of its children, all nodes in the BVH can be efficiently refitted by iterating over the array in reversed order, as suggested by van den Bergen [18]. During this refitting we mark empty nodes and if a node has just one child and contains no objects, its skip index is set to the skip index of this child, otherwise to itself.

Since adjusting one node and inserting an object takes almost constant time, the creation of the complete BVH can be done in time linear to the number of nodes and objects in the BVH.

The update phase between two consecutive frames is basically a reconstruction. During the construction we saved every insertion index of all object nodes in the BVH. Using these indices, we can simply empty the hierarchy. Afterwards the objects are animated and the BVH gets rebuild as described before. Pseudocode for the update phase is given below.

Algorithm 2 Update Phase LBVH

```
1: empty hierarchy
2: for all objs o do
3:     animate o
4:     adjust obj node of o
5: end for
6: expand root node to enclose scene
7: for all objs o do
8:     calc index in BVH for o
9:     insert o into BVH
10: end for
11: refit hierarchy
```

4 Results and Discussion

To evaluate our methods, we have used a variety of test scenes. Here we present the results for three of them, which we think reveal both, the benefits and weaknesses of our strategies. All tests were performed on a PC with a 2GHz Intel Pentium Mobile processor and 512 MB of memory. The maximum allowed ray tree depth was two, i.e. one reflection and refraction was allowed. The predefined depth for the LBVH is set to 18. The memory requirements are approximately twice as much as for a uniform grid.

We compare the results of the approaches presented above to a complete rebuild of the hierarchy every frame using the method of Goldsmith and Salmon [5]. The rebuild is done only once per frame, without further shuffling of the objects, since rebuilding the acceleration structure more than once is not feasible, if we want a fair comparison of the resulting update times. A simple refit of the BVH is not included in our statistics because of the drastic increase in ray tracing time for most of the scenes. The average (avg) timings for the update phase (up), ray tracing phase (rt) as well as the average speed-up achieved for the test scenes are presented in table 1.

Please note, that we assume no knowledge about possible movements of the objects. Therefore the kitchen scene e.g. is handled as a scene with about $110k$ dynamic objects (even though only 5 do move), and not as a scene consisting of almost only static objects and a few moving ones.

The first test scene is the kitchen scene from the BART benchmark suite [10]. Only a small toy car, consisting of 5 dynamic objects, is animated in an otherwise static surrounding. Test results are given in table 1 and Fig. 7, which shows a comparison of the ray tracing phase in the upper left graph and the update phase in the upper right graph.

The timings in the ray tracing phase between the complete rebuild and the Dyn. G&S method are almost equal. But when comparing the update timings, we achieved a dramatic decrease compared to a complete rebuild by more than two orders of magnitude. Showing the advantage of this method for scenes with small amounts of movement.

The update time for the LBVH also shows a decrease by more than an order of magnitude and almost constant update times, even though the ray tracing time almost doubled. A closer statistical analysis showed that this is due to the so-called "teapot in the stadium" problem. The scene consists of many very small objects. Due to the predefined depth of the hierarchy, the average number of primitives per leaf node is 100, with a maximum value of 1339. Therefore a two level approach, as presented in [11] and [20], should be used with the LBVH to avoid this problem.

The second test scene, the BART museum scene shows a museum room with a deforming piece of art in the middle, with mostly unstructured, random movement. The time spent in the update phase is reduced by roughly a factor of six to 15. The fact that the ray tracing time of the Dyn. G&S

Kitchen	G&S	Dyn. G&S	LBVH
avg. up	1.759s	0.017s	0.157s
avg. rt	6.142s	6.082s	11.771s
speed-up up	1.0	103.471	11.204
speed-up rt	1.0	1.010	0.522
speed-up tot	1.0	1.295	0.662
#tris 110k		resolution 300×225	

Museum	G&S	Dyn. G&S	LBVH
avg. up	1.933s	0.315s	0.125s
avg. rt	11.839s	7.950s	10.323s
speed-up up	1.0	6.137	15.464
speed-up rt	1.0	1.489	1.147
speed-up tot	1.0	1.666	1.318
#tris 76k		resolution 800×640	

Falling tris	G&S	Dyn. G&S	LBVH
avg. up	7.478s	0.907s	0.404s
avg. rt	22.298s	11.420s	3.182s
speed-up up	1.0	8.245	18.510
speed-up rt	1.0	1.953	7.008
speed-up tot	1.0	2.416	8.303
#tris 149k		resolution 512×512	

Table 1: Performance measurements from the three test scenes.

method does not exceed the ray tracing time for a BVH after a complete rebuild verifies our assumptions made in Sect. 3.1. In addition even a decrease in the time needed for the ray tracing phase is achieved, compared to a complete rebuild. In the case of the Dyn. G&S method, this is due to the possibility to rebuild the BVH several times in the beginning and shows the quality of our update routine. A local update of the acceleration data structure is sufficient to preserve the quality of a BVH.

The LBVH shows also good results in this test. It is not only able to ray trace the animation faster than the complete rebuild method, but it also takes only a fraction of the time needed in the update phase.

The last test scene consists of triangle patches, randomly assorted in a plane parallel to the xz-plane. During the animation, the triangles start falling from the ceiling at random times, speed and directions. The high number of animated primitives, as well as the highly changing object distribution, stresses our methods.

To avoid the advantage of the Dyn. G&S method of exploiting spatial

locality too much, the amount of frames is reduced to eleven. Therefore, the triangles move rather fast through the scene compared to the complete scene extents.

The arrangements of the primitives and the fact that they are all of uniform size leads to a relatively unbalanced tree, when using the Goldsmith and Salmon technique, while probably the best possible subdivision in this scene is simply always along the middle of the longest axis. As we rebuild the initial BVH for the Dyn. G&S method several times, it has a much better initial stand. Unfortunately, this is not possible for the complete rebuild method as it would take too long during the update phase.

As one can see, the time needed for the update phase of the LBVH is almost constant throughout the whole scene. It also has the best ray tracing time, which is due to the uniform size of the objects, which allows for a very good spatial partitioning.

The update time for both of our methods is beneath one second on average, theoretically allowing for interactive frame rates.

5 Conclusion and Future Work

In this report, we presented two methods for updating BVHs for ray tracing dynamic scenes. We have shown that the use of these two methods can greatly decrease the time needed in the update phase, compared to a complete rebuild. Speed-ups up to a factor of 103 in the update phase have been witnessed. This allows for much better overall performance, especially when using multiprocessor machines or techniques like frameless rendering. To our knowledge, this is the first time that the BART museum scene, at its highest complexity level (>64K moving objects), could be rendered and updated at interactive rates without using predefined hierarchies, at least theoretically.

The LBVH is also useful for time-critical ray tracing applications, as the update phase is almost constant in all tested scenes. Further optimizations are possible, in case of static scene extents, because of the missing expansion of the root node, this would also allow for calculating the insertion indices and building of the BVH in parallel. Moreover, approximate insertion levels can be precomputed if scaling is not allowed.

While the Dyn. G&S approach seems to be superior to a complete rebuild in all test cases, the LBVH suffers from its predefined depth in larger scenes with a highly non-uniform object distribution. As future work, we are planning to combine both strategies into a single algorithm to circumvent the "teapot in the stadium" problem of the LBVH and make it more suitable for a larger variety of scenes.

References

[1] G. Bishop, H. Fuchs, L. McMillan and E.J. Scher Zagier. Frameless Rendering: Double Buffering Considered Harmful. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques,* 1994, 175–176

[2] A. Dayal, C. Woolley, B. Watson and D.P. Luebke. Adaptive Frameless Rendering. In *Rendering Techniques,* 2005, 265–275

[3] M. Geimer. Interaktives Ray Tracing. In *PhD Thesis, University of Koblenz-Landau*, 2005

[4] A.S. Glassner. Spacetime Ray Tracing for Animation. In *IEEE Computer Graphics and Applications,* 8(2): 60–70, 1988

[5] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. In *IEEE Computer Graphics and Applications,* 7(5): 14–20, 1987

[6] J. Guenther, H. Friedrich, I. Wald, H.-P. Seidel and P. Slusallek Ray Tracing Animated Scenes using Motion Decomposition In *Proceedings of Eurographics,* 2006

[7] V. Isler, C.Aykanat and B. Oezguç. An Efficient Parallel Spatial Subdivision Algorithm for Parallel Ray Tracing Complex Scenes. In *First Bilkent Computer Graphics Conference, ATARV-93,* 1993, 121–134

[8] C. Lauterbach, S.-E. Yoon, D. Tuft and D. Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill*, 2006

[9] T. Larsson and T. Akenine-Moeller. Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models. In *Technical Report, MRTC Maelardalen Real-Time Research Centre, Maelardalen University*, 2003

[10] J. Lext, U. Assarsson and T. Moeller. A Benchmark for Animated Ray Tracing. In *IEEE Computer Graphics and Applications,* (21)2: 22–31, 2001

[11] J. Lext and T. Akenine-Moeller. Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 - Short Presentations,* 2001, 311–318

[12] J.D. MacDonald and K. Booth. Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface,* 1989, 152–163

[13] M.D.J. McNeill, B.C. Shah, M-P. Hébert, P.F. Lister and R.L. Grimsdale. Performance of Space Subdivision Techniques in Ray Tracing. In *Computer Graphics Forum,* 11(4): 213–220, 1992

[14] S. Parker, W. Martin, P.J. Sloan, P. Shirley, B. Smits and C. Hansen. Interactive Ray Tracing. In *Symposium on Interactive 3D Graphics,* 1999, 119–126

[15] E. Reinhard, B. Smits and C. Hansen. Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the 11th Eurographics Workshop on Rendering,* 2000, 299–306

[16] A. Reshetov, A. Soupikov and J. Hurley. Multi-level Ray Tracing Algorithm. In *ACM Transactions on Graphics,* 24(3): 1176–1185, 2005.

[17] T. Ulrich. Loose Octrees. In *Game Programming Gems,* 1: 434–442, 2000

[18] G. van den Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. In *Journal of Graphic Tools,*(2)4: 1–13, 1997

[19] I. Wald, C. Benthin, M. Wagner and P. Slusallek. Interactive Rendering with Coherent Ray Tracing. In *Computer Graphics Forum*, 2001, 153–164.

[20] I. Wald, C. Benthin and P. Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG),* 2003, 77–86

[21] Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. I. Wald, S. Boulos and P. Shirley. In *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-023 (conditionally accepted at ACM Transactions on Graphics),* 2006

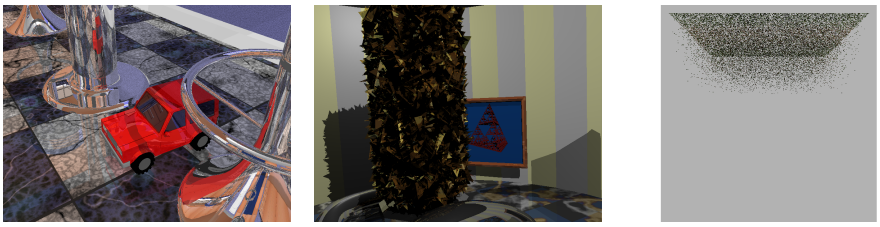

Figure 6:
Sample images from the three test scenes, Kitchen (left), Museum (middle) and Falling Triangles (right).

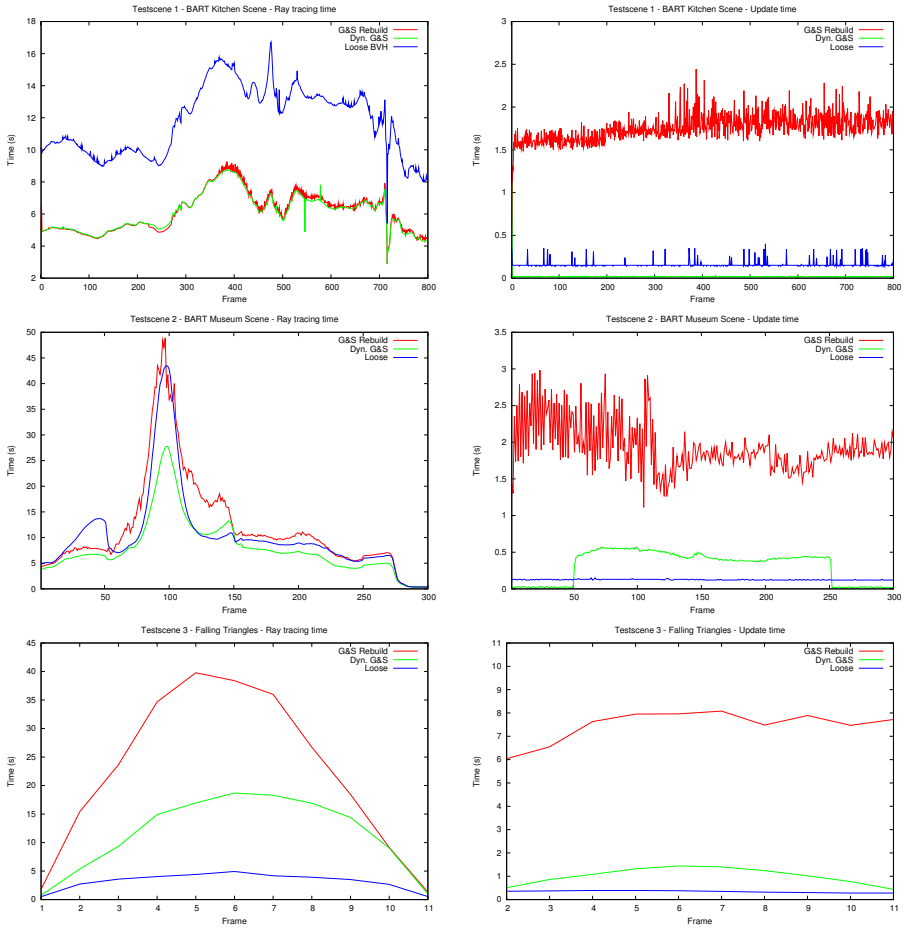

Figure 7: Test results for the three test scenes: Kitchen (top), Museum (middle), Falling Triangles (bottom). Left: Time spent in the ray tracing phase. Right: Time spent in the update phase